CIS 130 - Intro to Programming - Fall 2005
Homework Assignment #10 - **REVISED 11-23, fixed typo in Problem 3bab**

Homework #10:
HW #10 PART 1 is due by **12:00 NOON** on **Wednesday, November 30, 2005;**
HW #10 PART 2 is due by **12:00 NOON** on **Friday, December 2, 2005**

There is NO Week 13 Lab Exercise.

HOMEWORK #10

You are to work **individually** on this assignment.

PART 1: (30% of the HW #10 grade)
Complete problems #1 and #2 below, and submit the files they mention (using **~st10/130submit**) by **12:00 noon** on **Wednesday, November 30th** to receive **any** credit for Part 1 of HW #10.

PART 2: (70% of HW #10 grade)
Complete problems #3 - #6 below. When you are ready, you must submit the files specified in each problem (using **~st10/130submit**) by **12:00 noon** on **Friday, December 2nd** to receive **any** credit for Part 2 of HW #10.

1. Quick review: you should be quite familiar with a count-controlled while loop at this point:

```
const int NUM_DESIRED = 15;
int ct;
ct = 0;

while (ct < NUM_DESIRED)
{
    cout << ct << endl;
    ct = ct + 1;
}
```

And, from the lecture and readings on for-loops, you will hopefully not be surprised that below is the (mostly) equivalent for-loop implementing the same thing:

```
const int NUM_DESIRED = 15;

for ( int ct = 0; ct < NUM_DESIRED; ct = ct + 1)
{
    cout << ct << endl;
}
```

As a simple warm-up, consider **how_many_spam_msgs** from the Week "12" lab exercise:

```
// Contract: how_many_spam_msgs: int -> void
//
// Purpose: Print the message "I LIKE SPAM!" to the screen <num_times>
//          times, once per line.
//
```

```
// Examples: how_many_spam_msgs(3) should cause the following to be
//          written to the screen:
// I like Spam!
// I like Spam!
// I like Spam!
//
// by: Sharon M. Tuttle
// last modified: 11-3-05

#include <iostream>
using namespace std;

void how_many_spam_msgs(int num_times)
{
    int ct;

    ct = 0;

    while (ct < num_times)
    {
        cout << "I LIKE SPAM!" << endl;
        ct = ct + 1;
    }
}
```

Modify this so that is now uses a properly-structured **for-loop** instead of a count-controlled while-loop.

Try out your new version of **how_many_spam_msgs** by running it from the main function in **spam3.cpp**; when you are satisfied with it, submit **how_many_spam_msgs.cpp** using **~st10/130submit**.

2. You will recall that we discussed 1-dimensional **arrays** as well --- so, an array of 5 integer gerbil weights could be declared using:

```
const int NUM_GERBILS = 5;
int gerbil_wts[NUM_GERBILS];
```

And, you should also recall that `gerbil_wts[3]`, for example, then refers to the **fourth** gerbil weight in this array --- the first is at `gerbil_wt[0]`, the second is at `gerbil_wt[1]`, and so on.

A tidbit I didn't mention yet, but that you may find useful: in C++, when you declare an array, you can immediately set it to a set of values by writing the desired values in curly braces, separated by commas --- for example,

```
int gerbil_wts[NUM_GERBILS] = {10, 8, 6, 14, 7}
```

This can be especially useful when you want to set up an array for testing. (In other cases, you might ask the user for values interactively, using a for-loop and setting **my_array[ct]** to be the latest value, or you might read values from a file into an array, or somehow compute/generate/etc. the desired values and then store them in an array.)

As you can imagine, there might be times when you'd like to write a function that has an array as one of its parameters --- but, you'd hate to have to write one version for a 10-element array, another for a 15-element array, etc. But, in C++, you don't have to; you can declare an array with NO number of elements indicated, and then pass as its argument an array of ANY size. (How C++ actually gets away with this is a topic for CIS 230...)

The slight catch to this convenience is that you cannot "look" at a C++ array and "know" how big it is --- you have to be told. So, almost always, when a function takes an array as an argument, it also takes another argument also, an integer, that represents how many elements are in that array.

Here's an example: this function, **print_array**, takes an array of integers and its size as its two arguments. It simply prints each element of the array on its own line (notice how, in the contract, the type of an integer array parameter is written as **int[]**; in the header, the array parameter declaration is written like a "normal" array declaration, except with no size in the [ ]):

```
// Contract: int[] int -> void
// Purpose: print each of the <num_elements> elements in the array
//          <my_array> to the screen, each on its own line.
// Examples: if int quiz_grades[3] == {70, 100, 3}, then
//           print_array(quiz_grades, 3) would cause:
// 70
// 100
// 3
//           ...to be written to the screen

#include <iostream>
using namespace std;

void print_array(int my_array[], int num_elements)
{
    for (int ct = 0; ct < num_elements; ct = ct + 1)
    {
        cout << my_array[ct] << endl;
    }
}
```

WITH that set up --- write a main function in a file named **babytest.cpp**. It should run the example mentioned in print_array's opening comment block, **AND** it should also run another example call to print_array of your choice (on another array that you declare and fill). Before each call to print_array, include a cout that describes what you *should* be about to see (describe the expected value, since == won't work for non-returned results); for example,

```
cout << "About to call print_array(quiz_grades, 3) --- should see"
     << " 70, 100, 3, each on its own line" << endl;
print_array(quiz_grades, 3);
```

Remember that you'll have to declare and set up the example arrays for these calls! AND, for testing mains like this, I'm going to say that it is all right to use literals for the little test arrays' sizes. (But in a non-testing situation, you generally should use either parameters, named constants, or local variables for array sizes,

picking whichever is most appropriate.)

Test and debug your **babytest.cpp**; when you are satisfied with it, submit your **babytest.cpp** using **~st10/130submit**.

TYPO FIXED 11-23-05

3. And, let's do an example that involves the **char** type we discussed in lecture. (Remember, a char literal is enclosed in single quotes, and it is one character --- `'5'` is the *character* literal 5 (not the integer literal, which would be written `5` , nor the double literal, which could be written `5.0` . And `'a'` is the character literal **a**, `' '` is the character literal consisting of a blank character, etc.

Write a function **write_line** that expects **two** parameters: a character, and an integer. Its purpose is to write that character to the screen that many times, followed by a newline. (That is, **write_line('f', 4)** should cause the following to be printed to the screen:

```
ffff
```

One additional requirement: you are required to make appropriate use of a **for-loop** in **write_line**.

Even though this is interactive, you need to include examples in its opening comment block --- follow **print_array**'s example above to see an example of how this should be done for such a function. Include at least 3 examples with different characters and lengths.

To test this, write a main function in a file named **test_write_line.cpp**. It should run the examples mentioned in **write_line**'s opening comment block; before each, include a cout that describes what you *should* be about to see (since, again, == won't work with non-returned results). For example,

```
cout << "About to call write_line('f', 4) --- should see:" << endl
     << "ffff" << endl;
cout << "Do see: " << endl;
write_line('f', 4);
```

Again, note that, <u>for a testing main</u>, we're saying that using literals instead of named constants is acceptable.

Test and debug your **write_line.cpp** and **test_write_line.cpp**; when you are satisfied with them, create an example output file using:

```
test_write_line > test_write_line.out
```

Submit your **write_line.cpp**, **test_write_line.cpp**, and **test_write_line.out** using **~st10/130submit**. (I won't require you turn in **write_line.h** this time; you'll obviously have to have created it, though!)

4. What if you wanted to write a "box" of some character to the screen, instead of just a single line?

Write a function **write_box**; it should expect **three** parameters, a character, the height of the desired box, and the width of the desired box. (For example, **write_box('X', 3, 5)** should result in:

```
XXXXX
XXXXX
XXXXX
```

...being written to the screen.

Note the following requirements:
* your solution must call **write_line** appropriately;
* your solution must use a **for-loop** appropriately;
* include at least 2 examples in its opening comment block.

To test this, write a main function in a file named **test_write_box.cpp**. It should run the examples mentioned in **write_box**'s opening comment block; before each, include a cout that describes what you *should* be about to see (since, again, == won't work with non-returned results). For example,

```
cout << "About to call write_box('X', 3, 5) --- should see:" << endl
     << "a box of X's 3 X's high and 5 X's wide" << endl;
cout << "Do see: " << endl;
write_box('X', 3, 5);
```

Again, note that, <u>for a testing main</u>, we're saying that using literals instead of named constants is acceptable.

Test and debug your **write_box.cpp** and **test_write_box.cpp**; when you are satisfied with them, create an example output file using:

```
test_write_box > test_write_box.out
```

Submit your **write_box.cpp**, **test_write_box.cpp**, and **test_write_box.out** using **~st10/130submit**. (I won't require you turn in **write_box.h** this time; you'll obviously have to have created it, though!)

5. A bit more array practice is needed, however. So, let's start by noting that a simple bar chart could be created by putting a row of X's equal to each value in a set of values --- for example, for {3, 5, 2},

```
XXX
XXXXX
XX
```

Write a function **baby_bar**. It takes an integer array and the number of elements in that array as parameters; it prints a "baby bar chart" like that above to the screen, for each row printing the number of X's equal to the corresponding value in the array. For example, if

```
int my_test_array[5] = {1, 3, 5, 2, 7}
```

...then **`baby_bar(my_test_array, 5);`** should result in the following being printed to the screen:

```
X
XXX
XXXXX
XX
XXXXXXX
```

Additional requirements from **baby_bar**:
* your solution must call **write_line** appropriately;
* your solution must use a **for-loop** appropriately;

* include at least 1 example in its opening comment block.

To test this, write a main function in a file named **test_baby_bar.cpp**. It should run the example(s) mentioned in **baby_bar**'s opening comment block; before each, include a cout that describes what you *should* be about to see (since, again, == won't work with non-returned results). For example,

```
cout << "About to call baby_bar(my_test_array, 3) --- should see:" << endl
     << "1 X, then 3 X's, then 5 X's, then 2 X's, then 7 X's" << endl;
cout << "Do see: " << endl;
baby_bar(my_test_array, 3)
```

Again, note that, for a testing main, we're saying that using literals instead of named constants is acceptable.

Test and debug your **baby_bar.cpp** and **test_baby_bar.cpp**; when you are satisfied with them, create an example output file using:

```
test_baby_bar > test_baby_bar.out
```

Submit your **baby_bar.cpp**, **test_baby_bar.cpp**, and **test_baby_bar.out** using **~st10/130submit**. (I won't require you turn in **baby_bar.h** this time; you'll obviously have to have created it, though!)

6. Finally, I wouldn't want you to think that an array cannot be involved in a "pure"-style function; it certainly can be! Consider (and write) a function **num_too_big**; it could take an array of double values, the number of elements in that array, and an upper bound as its parameters, and return the number of elements in that array that are strictly greater than the given upper bound. For example,
for `double my_vals[8] = {3.4, 1, 3, 8, 20, 12.7, 2.01, 13.3};`

```
num_too_big(my_vals, 8, 10) == 3
num_too_big(my_vals, 8, 500) == 0
```

Additional requirements:
* **num_too_big** must make appropriate use of a **for-loop**.
* Include at least 3 examples in its opening comment block, including at least one for which all of the array values are less than the upper bound given.

You can write a testing main for this; do so, in file **test_num_in_excess.cpp**. Fill an array as needed to run the examples in **num_too_big**'s opening comment block, print a line to the screen saying that you are testing **num_too_big** and that 1's == passed and 0's == failed, and then print out the results of comparing the results of calling **num_too_big** with their expected results.

Again, note that, for a testing main, we're saying that using literals instead of named constants is acceptable.

Test and debug your **num_too_big.cpp** and **test_num_too_big.cpp**; when you are satisfied with them, create an example output file using:

```
test_num_too_big > test_num_too_big.out
```

Submit your **num_too_big.cpp**, **test_num_too_big.cpp**, and **test_num_too_big.out** using **~st10/130submit**. (I won't require you turn in **num_too_big.h** this time; you'll obviously have to have

created it, though!)